C++ Second Exam Review

Contents :

Loops :

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

```
while (expression) statement
```

and its functionality is simply to repeat statement while the condition set in expression is true.
For example, we are going to make a program to countdown using a while-loop:

```
// custom countdown using while

#include <iostream>
using namespace std;

int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "FIRE!\n";
  return 0;
}
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

When the program starts the user is prompted to insert a starting number for the countdown. Then the `while` loop begins, if the value entered by the user fulfills the condition `n>0` (that `n` is greater than zero) the block that follows the condition will be executed and repeated while the condition (`n>0`) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. User assigns a value to n
2. The while condition is checked (n>0). At this point there are two possibilities:
   * condition is true: statement is executed (to step 3)
   * condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   cout << n << ", ";
   --n;
   (prints the value of n on the screen and decreases n by 1)
4. End of block. Return automatically to step 2
5. Continue the program right after the block: print FIRE! and end program.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included `--n;` that decreases the value of the variable that is being evaluated in the condition (`n`) by one - this will eventually make the condition (`n>0`) to become false after a certain number of loop iterations: to be more specific, when `n` becomes `0`, that is where our while-loop and our countdown end.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

The do-while loop

Its format is:

```
do statement while (condition);
```

Its functionality is exactly the same as the while loop, except that `condition` in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of `statement` even if `condition` is never fulfilled. For example, the following example program echoes any number you enter until you enter `0`.

```
// number echoer

#include <iostream>
using namespace std;

int main ()
{
  unsigned long n;
  do {
    cout << "Enter number (0 to end): ";
    cin >> n;
    cout << "You entered: " << n << "\n";
  } while (n != 0);
  return 0;
}
```

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined **within** the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value `0` in the previous example you can be prompted for more numbers forever.

The for loop

Its format is:

```
for (initialization; condition; increase) statement;
```

and its main function is to repeat `statement` while `condition` remains true, like the while loop. But in addition, the `for`loop provides specific locations to contain an `initialization` statement and an `increase` statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

1. initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3. statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

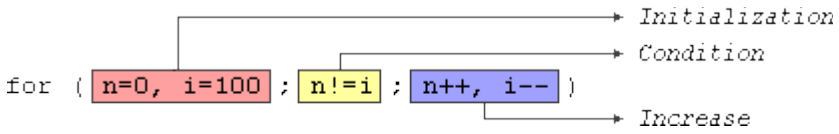
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The `initialization` and `increase` fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: `for (;n<10;)` if we wanted to specify no initialization and no increase; or `for (;n<10;n++)` if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (`,`) we can specify more than one expression in any of the fields included in a `for`loop, like in `initialization`, for example. The comma operator (`,`) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
   // whatever here...
}
```

This loop will execute for 50 times if neither `n` or `i` are modified within the loop:



`n` starts with a value of `0`, and `i` with `100`, the condition is `n!=i` (that `n` is not equal to `i`). Because `n` is increased by one and `i` decreased by one, the loop's condition will become false after the 50th loop, when both `n` and `i` will be equal to `50`.

The break statement

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end :

```
// break loop example

#include <iostream>
using namespace std;

int main ()
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!
```

Predefined Functions :

Function	Header File	Purpose	Parameter Type	Return Type
abs(x)	<cstdlib>	returns the absolute value of its argument	int x	int result
fabs(x)	<cmath>	returns the absolute value of its argument	double x	double result

sin(x)	<cmath>	returns the sine of angle x (radians) *	double x	double result
cos(x)	<cmath>	returns the cosine of angle x (radians) *	double x	double result
exp(x)	<cmath>	returns (e^x) where e = 2.718	double x	double result
ceil(x)	<cmath>	returns the smallest whole number that is not less than x	double x	double result
sqrt(x)	<cmath>	Returns the square root of x; x must be positive	double x	double result
floor(x)	<cmath>	returns the largest whole number that is not greater than x	double x	double result
pow(x,y)	<cmath>	If (x^y)is a negative number, y must be a whole number	double x	double result
tolower(x)	<cctype>	returns the lower case value of x	int x	int result
toupper(x)	<cctype>	returns the upper case value of x	int x	int result

* radians = (angle x 3.14)/180

User-Defined Functions :

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

```
type name ( parameter1, parameter2, ...) { statements }
```

where:

- type is the data type specifier of the data returned by the function.
- name is the identifier by which it will be possible to call the function.
- parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is a block of statements surrounded by braces { }.

Here you have the first function example:

```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return (r);
```

```
The result is 8
```

```
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution by the `main` function. So we will begin there.

We can see how the `main` function begins by declaring the variable `z` of type `int`. Right after that, we see a call to a function called `addition`. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:

```
int addition (int a, int b)
               ↑      ↑
z = addition (  5  ,   3  );
```

The parameters and arguments have a clear correspondence. Within the `main` function we called to `addition` passing two values: 5 and 3, that correspond to the `int a` and `int b` parameters declared for function addition.

At the point at which the function is called from within `main`, the control is lost by `main` and passed to function `addition`. The value of both arguments passed in the call (5 and 3) are copied to the local variables `int a` and `int b` within the function.

Function `addition` declares another local variable (`int r`), and by means of the expression `r=a+b`, it assigns to `r` the result of a plus b. Because the actual parameters passed for a and b are 5 and 3 respectively, the result is 8.

The following line of code:

```
return (r);
```

finalizes function `addition`, and returns the control back to the function that called it in the first place (in this case,`main`). At this moment the program follows its regular course from the same point at which it was interrupted by the call to `addition`. But additionally, because the `return` statement in function `addition` specified a value: the content of variable `r` (`return (r);`), which at that moment had a value of 8. This value becomes the value of evaluating the function call.

```
int addition (int a, int b)
 ↓8
z = addition (  5  ,   3  );
```

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable zwill be set to the value returned by `addition (5, 3)`, that is 8. To explain it another way, you can imagine that the call to a function (`addition (5,3)`) is literally replaced by the value it returns (8).

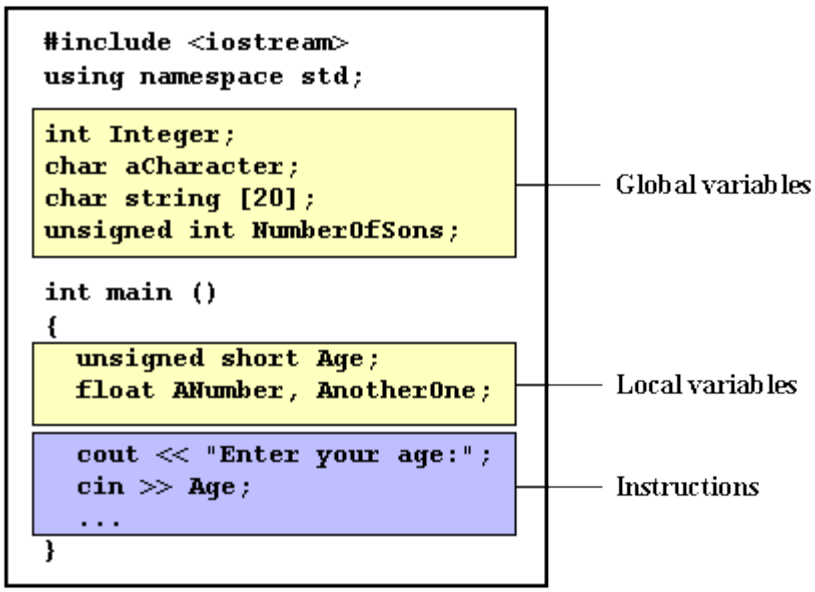
The following line of code in main is:

```
cout << "The result is " << z;
```

That, as you may already expect, produces the printing of the result on the screen.

Scope of variables

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variables `a`, `b` or `r` directly in function `main` since they were variables local to function `addition`. Also, it would have been impossible to use the variable `z` directly within function `addition`, since this was a variable local to the function `main`.



Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare global variables; These are visible from any point of the code, inside and outside all functions. In order to declare global variables you simply have to declare the variable outside any function or block; that means, directly in the body of the program.

And here is another example about functions:

```
1  // function example
2  #include <iostream>
3  using namespace std;
4
5  int subtraction (int a, int b)
6  {
7    int r;
8    r=a-b;
9    return (r);
10 }
11
12 int main ()
13 {
```

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

```
14   int x=5, y=3, z;
15   z = subtraction (7,2);
16   cout << "The first result is " << z << '\n';
17   cout << "The second result is " << subtraction (7,2) << '\n';
18   cout << "The third result is " << subtraction (x,y) << '\n';
19   z= 4 + subtraction (x,y);
20   cout << "The fourth result is " << z << '\n';
21   return 0;
22 }
```

In this case we have created a function called `subtraction`. The only thing that this function does is to subtract both passed parameters and to return the result.

Nevertheless, if we examine function `main` we will see that we have made several calls to function `subtraction`. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to fully understand these examples you must consider once again that a call to a function could be replaced by the value that the function call itself is going to return. For example, the first case (that you should already know because it is the same pattern that we have used in previous examples):

```
1 z = subtraction (7,2);
2 cout << "The first result is " << z;
```

If we replace the function call by the value it returns (i.e., 5), we would have:

```
1 z = 5;
2 cout << "The first result is " << z;
```

As well as

```
cout << "The second result is " << subtraction (7,2);
```

has the same result as the previous call, but in this case we made the call to `subtraction` directly as an insertion parameter for `cout`. Simply consider that the result is the same as if we had written:

```
cout << "The second result is " << 5;
```

since 5 is the value returned by `subtraction (7,2)`.

In the case of:

```
cout << "The third result is " << subtraction (x,y);
```

The only new thing that we introduced is that the parameters of `subtraction` are variables instead of constants. That is perfectly valid. In this case the values passed to function `subtraction` are the values of x and y, that are 5 and 3respectively, giving 2 as result.

The fourth case is more of the same. Simply note that instead of:

```
z = 4 + subtraction (x,y);
```

we could have written:

```
z = subtraction (x,y) + 4;
```

with exactly the same result. I have switched places so you can see that the semicolon sign (;) goes at the end of the whole statement. It does not necessarily have to go right after the function call. The explanation might be once again that you imagine that a function can be replaced by its returned value:

```
z = 4 + 2;
z = 2 + 4;
```

Functions with no type. (The use of void)

If you remember the syntax of a function declaration:

`type name ( argument1, argument2 ...) statement`

you will see that the declaration begins with a `type`, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the `void` type specifier for the function. This is a special specifier that indicates absence of type.

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
  return 0;
}
```

```
I'm a function!
```

`void` can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function `printmessage` could have been declared as:

```
void printmessage (void)
{
  cout << "I'm a function!";
}
```

Although it is optional to specify `void` in the parameter list. In C++, a parameter list can simply be left blank

if we want a function with no parameters.

What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to `printmessage` is:

```
printmessage ();
```

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call would have been incorrect:

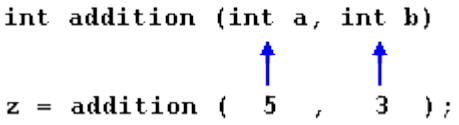
```
printmessage;
```

Arguments passed by value and by reference.

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function `addition` using the following code:

```
int x=5, y=3, z;
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.



This way, when the funct**Declaring functions**ion addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

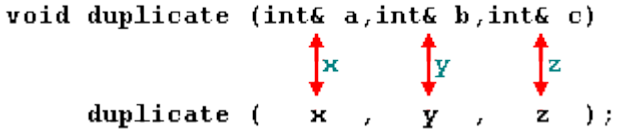
```
// passing parameters by reference
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
```

```
x=2, y=6, z=14
```

```
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

The first thing that should call your attention is that in the declaration of `duplicate` the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed*by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.



To explain it in another way, we associate `a`, `b` and `c` with the arguments passed on the function call (`x`, `y` and `z`) and any change that we do on `a` within the function will affect the value of `x` outside it. Any change that we do on `b` will affect `y`, and the same with `c` and `z`.

That is why our program's output, that shows the values stored in `x`, `y` and `z` after the call to `duplicate`, shows the values of all the three variables of `main` doubled.

If when declaring the following function:

```
void duplicate (int& a, int& b, int& c)
```

we had declared it this way:

```
void duplicate (int a, int b, int c)
```

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of `x`, `y` and `z` without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
// more than one returning value
#include <iostream>
using namespace std;

void prevnext (int x, int& prev, int& next)
{
  prev = x-1;
  next = x+1;
}

```

```
Previous=99, Next=101
```

```
int main ()
{
  int x=100, y, z;
  prevnext (x, y, z);
  cout << "Previous=" << y << ", Next=" << z;
  return 0;
}
```

Default values in parameters.

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12);
  cout << endl;
  cout << divide (20,4);
  return 0;
}
```

```
6
5
```

As we can see in the body of the program there are two calls to function `divide`. In the first one:

```
divide (12)
```

we have only specified one argument, but the function `divide` allows up to two. So the function `divide` has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with `int b=2`, not just `int b`). Therefore the result of this function call is 6(12/2).

In the second call:

```
divide (20,4)
```

there are two parameters, so the default value for `b` (`int b=2`) is ignored and `b` takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).

Overloaded functions

In C++ two different functions can have the same name if their parameter types or number are different.

That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:

```
// overloaded function
#include <iostream>
using namespace std;

int operate (int a, int b)
{
  return (a*b);
}

float operate (float a, float b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << operate (x,y);
  cout << "\n";
  cout << operate (n,m);
  cout << "\n";
  return 0;
}
```

```
10
2.5
```

In this case we have defined two functions with the same name, `operate`, but one of them accepts two parameters of type `int` and the other one accepts them of type `float`. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two `int` parameters in its prototype and if it is called with two floats it will call to the one which has two`float` parameters in its prototype.

In the first call to `operate` the two arguments passed are of type `int`, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type `float`, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to `operate` depends on the type of the arguments passed because the function has been *overloaded*.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Declaring functions

Until now, we have defined all of the functions before the first appearance of calls to them in the source code. These calls were generally in function `main` which we have always left at the end of the source code. If you try to repeat some of the examples of functions described so far, but placing the function `main` before any of the other functions that were called from within it, you will most likely obtain compiling errors. The reason is that to be able to call a function it must have been declared in some earlier point of the code, like we have done in all our examples.

But there is an alternative way to avoid writing the whole code of a function before it can be used in main or in some other function. This can be achieved by declaring just a prototype of the function before it is used, instead of the entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

Its form is:

```
type name ( argument_type1, argument_type2, ...);
```

It is identical to a function definition, except that it does not include the body of the function itself (i.e., the function statements that in normal definitions are enclosed in braces { }) and instead of that we end the prototype declaration with a mandatory semicolon (;).

The parameter enumeration does not need to include the identifiers, but only the type specifiers. The inclusion of a name for each parameter as in the function definition is optional in the prototype declaration. For example, we can declare a function called `protofunction` with two `int` parameters with any of the following declarations:

```
int protofunction (int first, int second);
int protofunction (int, int);
```

Anyway, including a name for each variable makes the prototype more legible.

```
// declaring functions prototypes
#include <iostream>
using namespace std;

void odd (int a);
void even (int a);

int main ()
{
  int i;
  do {
    cout << "Type a number (0 to exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int a)
{
  if ((a%2)!=0) cout << "Number is odd.\n";
  else even (a);
}

void even (int a)
{
  if ((a%2)==0) cout << "Number is even.\n";
  else odd (a);
}
```

```
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
Type a number (0 to exit): 1030
Number is even.
Type a number (0 to exit): 0
Number is even.
```

This example is indeed not an example of efficiency. I am sure that at this point you can already make a program with the same result, but using only half of the code lines that have been used in this example. Anyway this example illustrates how prototyping works. Moreover, in this concrete example the prototyping of at least one of the two functions is necessary in order to compile the code without errors.

The first things that we see are the declaration of functions odd and even:

```
void odd (int a);
void even (int a);
```

This allows these functions to be used before they are defined, for example, in `main`, which now is located where some people find it to be a more logical place for the start of a program: the beginning of the source code.

Anyway, the reason why this program needs at least one of the functions to be declared before it is defined is because inodd there is a call to even and in even there is a call to odd. If none of the two functions had been previously declared, a compilation error would happen, since either odd would not be visible from even (because it has still not been declared), oreven would not be visible from odd (for the same reason).

Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.

Suggested Problems :

1. Write a value-returning function, isVowel, that returns the value true if a given character is a vowel and otherwise returns false.

2. Write a program that prompts the user to input a sequence of characters and outputs the number of vowels (Use the function is vowel written in Q1)

3.Write a function, reverseDigit, that takes an integer as a parameter and returns the number with its digits reversed. For example, the value of reverseDigit(12345) is 54321; the value of reverseDigit(5600) is 65; the value of reverseDigit(-532) is -235.

5. Write a program that prompts the user to input a positive integer. It should then output a message indicating whether the number is prime number.

Problem Solutions :

1.

```
#include <iostream>
#include <cctype>
using namespace std;

bool isVowel(char c)
{
c = tolower(c);
if ((c == 'a') || (c == 'e') || (c == 'i') || (c == 'o') || ( c == 'u'))
return true;
else
return false ;
}

int main()
{
char ch;
cout << "Enter a character: ";
cin >> ch;

if (isVowel(ch))
 cout << "You entered a vowel\n";
else
 cout << "You entered a consonant\n";

}
```

2.

```
#include <iostream>
#include <cctype>
using namespace std;

bool isVowel(char c)
{
c = tolower(c);
if ((c == 'a') || (c == 'e') || (c == 'i') || (c == 'o') || ( c == 'u'))
```

```
return true;
else
return false ;
}

int main()
{
char s;
int count = 0;
cout << "Input as many characters as you want,\nif you want to stop the program enter space: ";

do
{
s = cin.get();
count += isVowel(s);
}
while(s != ' ');

cout <<"The number of Vowels is: "<< count<<endl;

}
```

3.

```
#include <iostream>
#include <cstdlib>
using namespace std;

int z = 1;
void reverseDigit()
{
int i,k=0;
cout<<"enter an int (under 10 digits): "<<endl;
cin>>i;
if (i < 0)
z = -1;
i = abs(i);
while(i>=9)
{
k+=i%10;
i=i/10;
```

```
k=k*10;
}
if(i!=0)
{
k=k+i;
}
cout << k*z;
}


int main()
{
reverseDigit();

return 0;
}
```

4.

```
#include <iostream>
using namespace std;
      int main()
       {

        int number,count=0;
        cout<<"ENTER NUMBER TO CHECK IT IS PRIME OR NOT (UNDER 10 DIGITS)\n ";
         cin>>number;
          for(int a=1;a<=number;a++)
             {
              if(number%a==0)
                 {
                count++;
                 }
             }
      if(count==2)
        {
         cout<<" PRIME NUMBER \n";
        }
      else
        {
         cout<<" NOT A PRIME NUMBER \n";
```

```
  }

return 0;
    }
```